A Result Language (OSrL) and Solver Option Language (OSoL) for Distributed Optimization

Robert Fourer and Jun Ma
Northwestern University

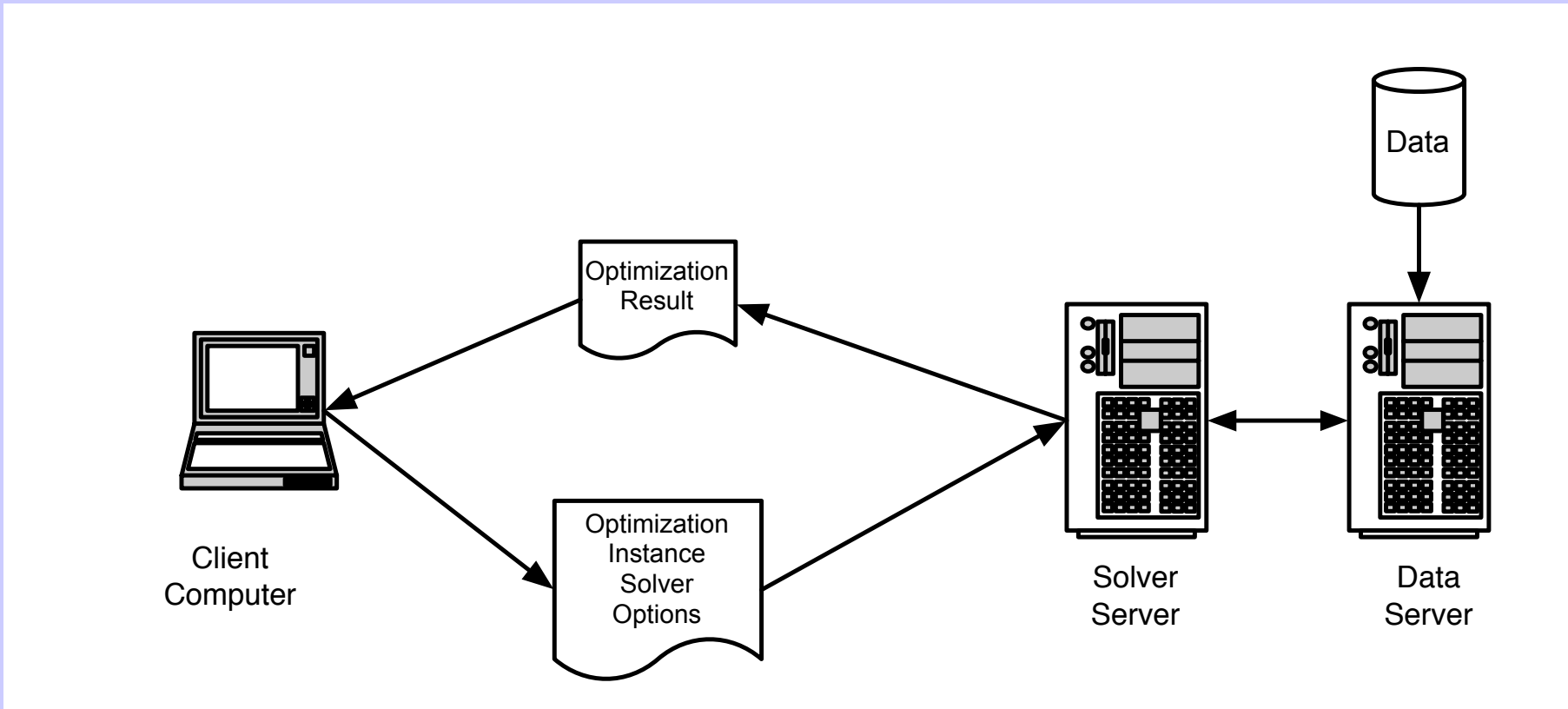
Kipp Martin
University of Chicago

November 8, 2006

Outline

Basic Philosophy

Scenario: a client machine with a modeling language/instance talks to a solver either locally or on a remote machine.



In general, the instance generation and instance solution are not part of the same process.

Basic Philosophy

Design Patterns: There should be four distinct parts to the **interface** between the problem and the solver.

- a problem instance interface API – create the instance and get information about the instance
- a solver option interface – an instance representation should be independent of solver options
- a problem result interface
- a problem modification interface

In each case there is a file representation and corresponding in-memory representation. This correspondence is very tight and not arbitrary as we see later.

Basic Philosophy

The file **testremote.config:**

```
-osil ../data/parincLinear.osil
-solver lindo
-osol ../data/demo.osol
-serviceLocation http://***/lindo/LindoSolverService.jws
-browser  /Applications/Firefox.app/Contents/MacOS/firefox
-osrl /Users/kmartin/temp/test2.osrl
-serviceMethod solve
```

The file **testlocal.config:**

```
-osil ../data/parincLinear.osil
-solver cbc
-browser  /Applications/Firefox.app/Contents/MacOS/firefox
-osol ../data/demo.osol
-serviceMethod solve
-osrl /Users/kmartin/temp/test2.osrl
```

Basic Philosophy

The *OSSolverService* executable implements our philosophy. For example,

OSSolverService -config testremote.config

OSSolverService -config testlocal.config

OSSolverService -config testlocal.config -solver clp

Basic Philosophy

For each standard there is an associated XML file representation (and associated schema) and corresponding in-memory representation.

File	**In Memory**
OSiL	OSInstance
OSrL	OSResult
OSosL	OSOption

OS Protocols: OSrL

```
<variables>
     <values>
          <var idx="0">539.984</var>
          <var idx="1">252.011</var>
     </values>
</variables>
<objectives>
     <values>
      <obj idx="-1">7667.94</obj>
     </values>
</objectives>
<dualValues>
<con idx="0">4.37457</con><con idx="1">0</con>
</dualValues>
```

The fact that the result is in XML has important implications. It is now easy to write XSLT (Extensible Stylesheet Language Transformation) stylesheets to transform the result into human readable HTML.

The OSrL Schema

Design Goal: *maximize flexibility in reporting optimization results but keep the design simple!*

With OSiL we try to be as encompassing and complete as possible. We try to represent all interesting optimization instances.

With OSrL we take a minimalist approach.

A linear program is a well defined entity, the solution of a linear program is not. We can't have a linear program without constraints; however, we can have a linear programming solution without reduced costs.

The OSrL Schema

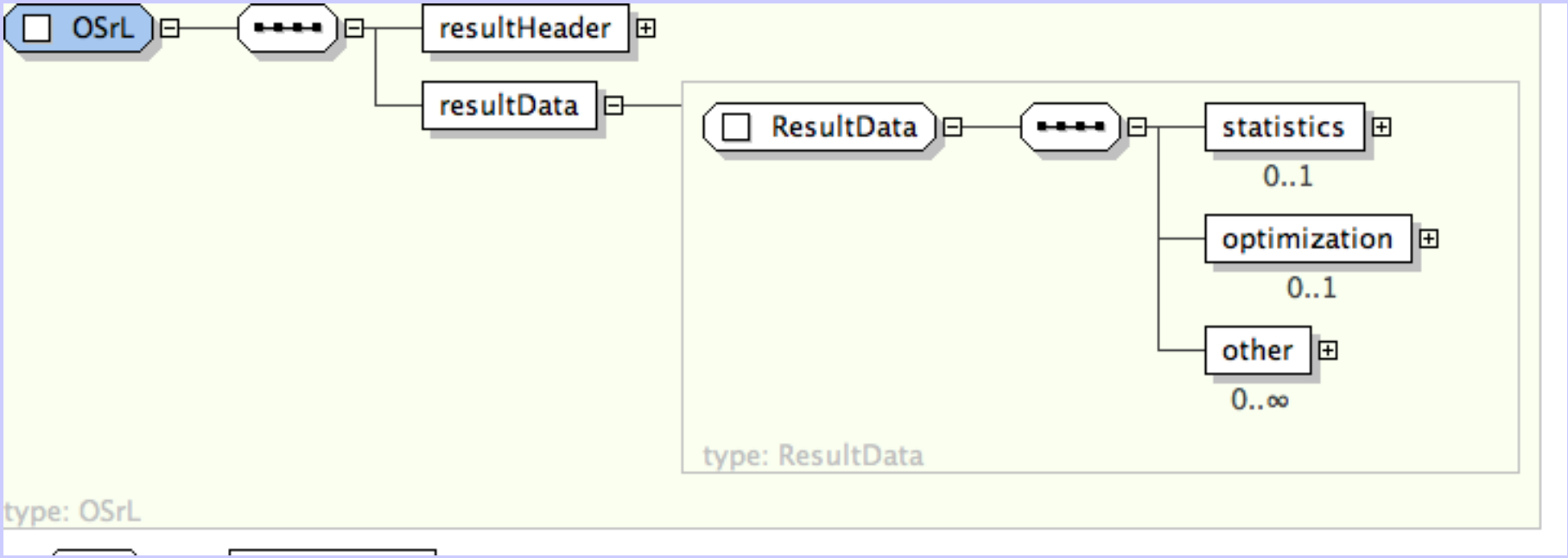
Design Goal: *maximize flexibility in reporting optimization results but keep the design simple!*

Different linear or nonlinear optimization codes may present their results in different formats, some may include more detail than others. That is fine, it is up to the solver developer. We provide that capability.

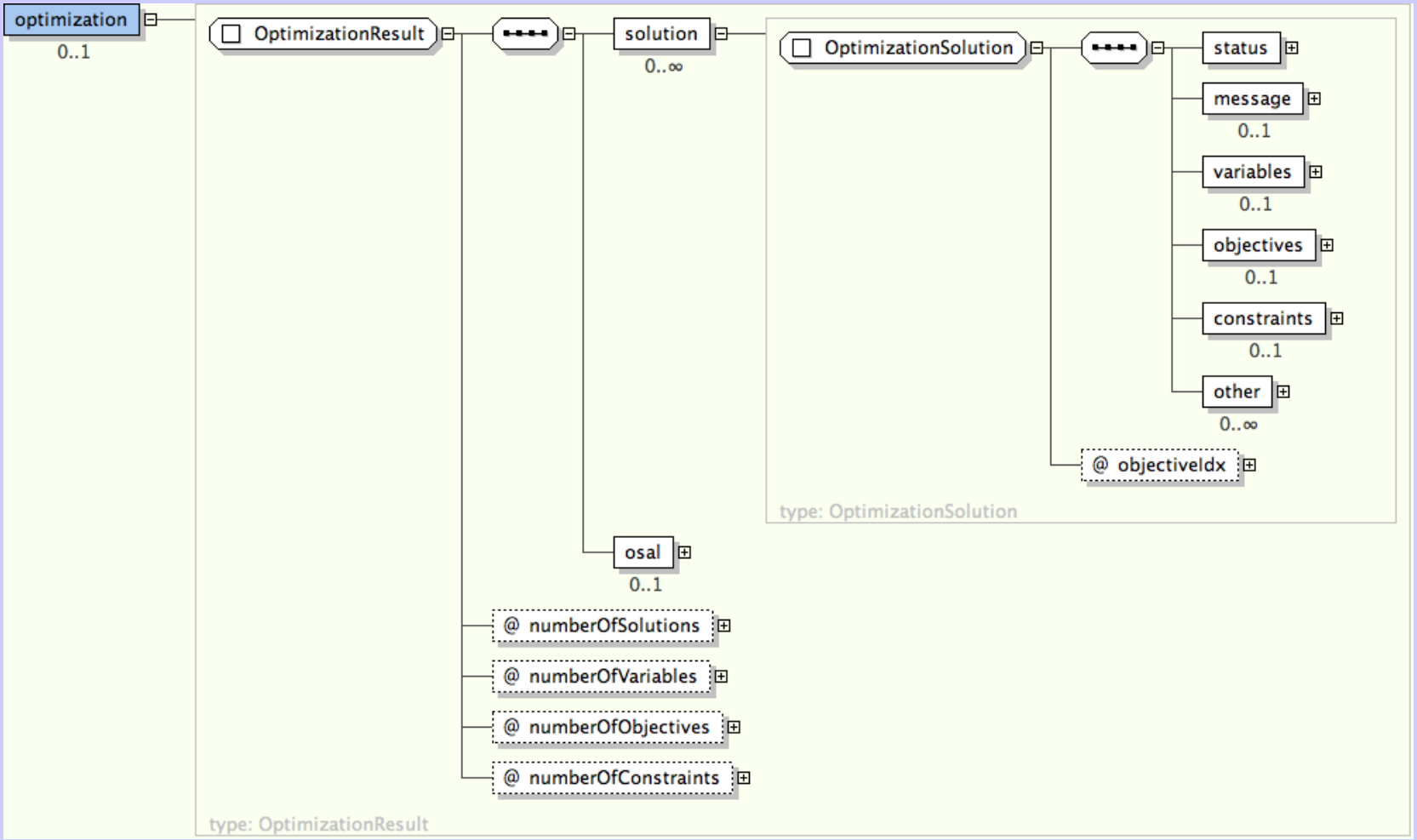
– BUT –

This will not work for representing problem instances. It would be very bad indeed to have numerous different formats for problem instances. This would make it hard for both modeling language developers and solver developers.

The OSrL Schema



The OSrL Schema



The OSrL Schema

The Result Header

```
<resultHeader>
   <generalStatus type="success"/>
   <serviceName>Solved with Coin Solver: clp</serviceName>
   <instanceName>Par Inc. </instanceName>
</resultHeader>
```

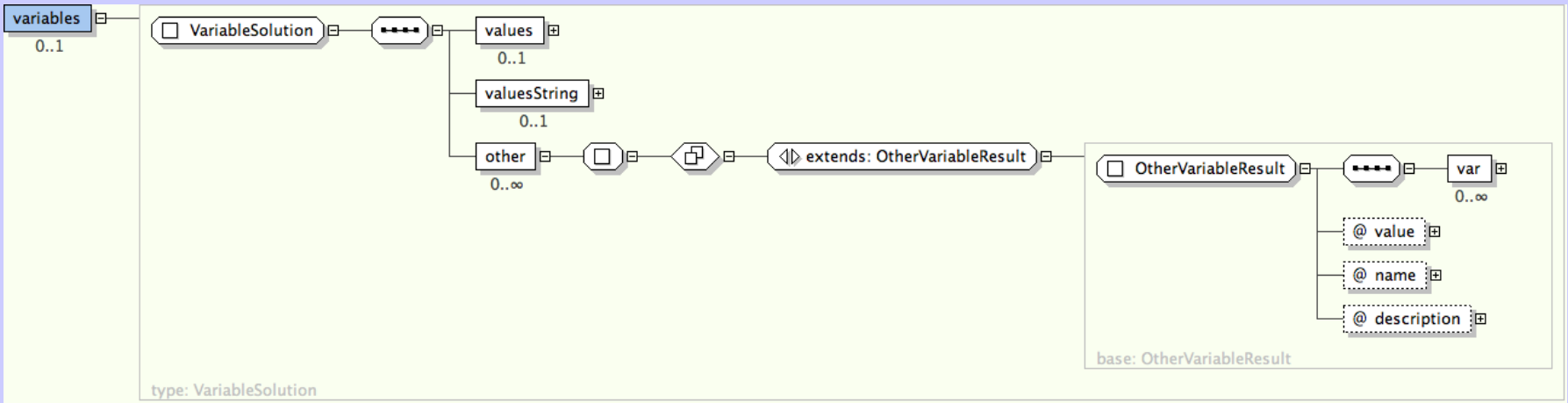
The OSrL Schema

The Result Data

```
<resultData>
<optimization numberOfSolutions="1" numberOfVariables="2" numberOfConstraints="4" numberOfObjectives="1">
<solution objectiveIdx="-1">
<status type="optimal"/>
<variables>
<values>
<var idx="0">539.984</var>
<var idx="1">252.011</var>
</values>
</variables>
<objectives>
<values>
<obj idx="-1">7667.94</obj>
</values>
</objectives>
<constraints>
<dualValues>
<con idx="0">4.37457</con>
<con idx="1">-0</con>
<con idx="2">6.9378</con>
<con idx="3">-0</con>
</dualValues>
</constraints>
</solution>
</optimization>
</resultData>
```

OS Protocols: OSrL

The use of the `<other>` element for variables.



OS Protocols: OSrL

Use the <variables> element for variables to represent reduced costs.

```
<xs:complexType name="VariableSolution">
   <xs:sequence>
      <xs:element name="values"
         type="VariableValues" minOccurs="0"/>
      <xs:element name="other" minOccurs="0"
         maxOccurs="unbounded">
         <xs:complexType>
         <xs:complexContent>
         <xs:extension base="OtherVariableResult"/>
         </xs:complexContent>
        </xs:complexType>
      </xs:element>
      </xs:sequence>
</xs:complexType>
```

OS Protocols: OSrL

Use the `VariableSolution` class for variables to represent reduced costs. The C++ code.

```
class VariableSolution{
public:
int numberOfOtherVariableResult;
VariableValues *values;
OtherVariableResult **other;
VariableSolution();
~VariableSolution();
};// class VariableSolution
```

OS Protocols: OSrL

Use the <OtherVariableResult> element for variables to represent reduced costs.

```
<xs:complexType name="OtherVariableResult">
<xs:sequence>
<xs:element name="var" type="OtherVarResult"
minOccurs="0" maxOccurs="unbounded"/>
</xs:sequence>
<xs:attribute name="name" type="xs:string" use="required"/>
<xs:attribute name="description" type="xs:string"
use="optional"/>
</xs:complexType>
```

OS Protocols: OSrL

Use the `OtherVariableResult` class for variables to represent reduced costs. The C++ code.

```
class OtherVariableResult {
public:
   std::string name;
   std::string description;
   OtherVarResult **var;
   OtherVariableResult();
   ~OtherVariableResult();
};//OtherVariableResult
```

OS Protocols: OSrL

Use the <other> element for variables to represent reduced costs.

```
<variables>
    <values>
    <var idx="0">539.984</var>
    <var idx="1">252.011</var>
    </values>
    <other name="reduced costs"
         description="the variable reduced costs">
    <var idx="0">0</var>
    <var idx="1">0</var>
    </other>
 </variables>
```

OS Protocols: OSrL

We could do exactly the same thing to represent basic variables.

```
 <other name="reduced costs"
     description="the variable reduced costs">
<var idx="5">32.57</var>
<var idx="7">100.99</var>
</other>
<other name="basic variables"
     description="list the variables in the basis">
<var idx="4">basic</var>
<var idx="9">basic</var>
</other>
```

OS Protocols: OSrL

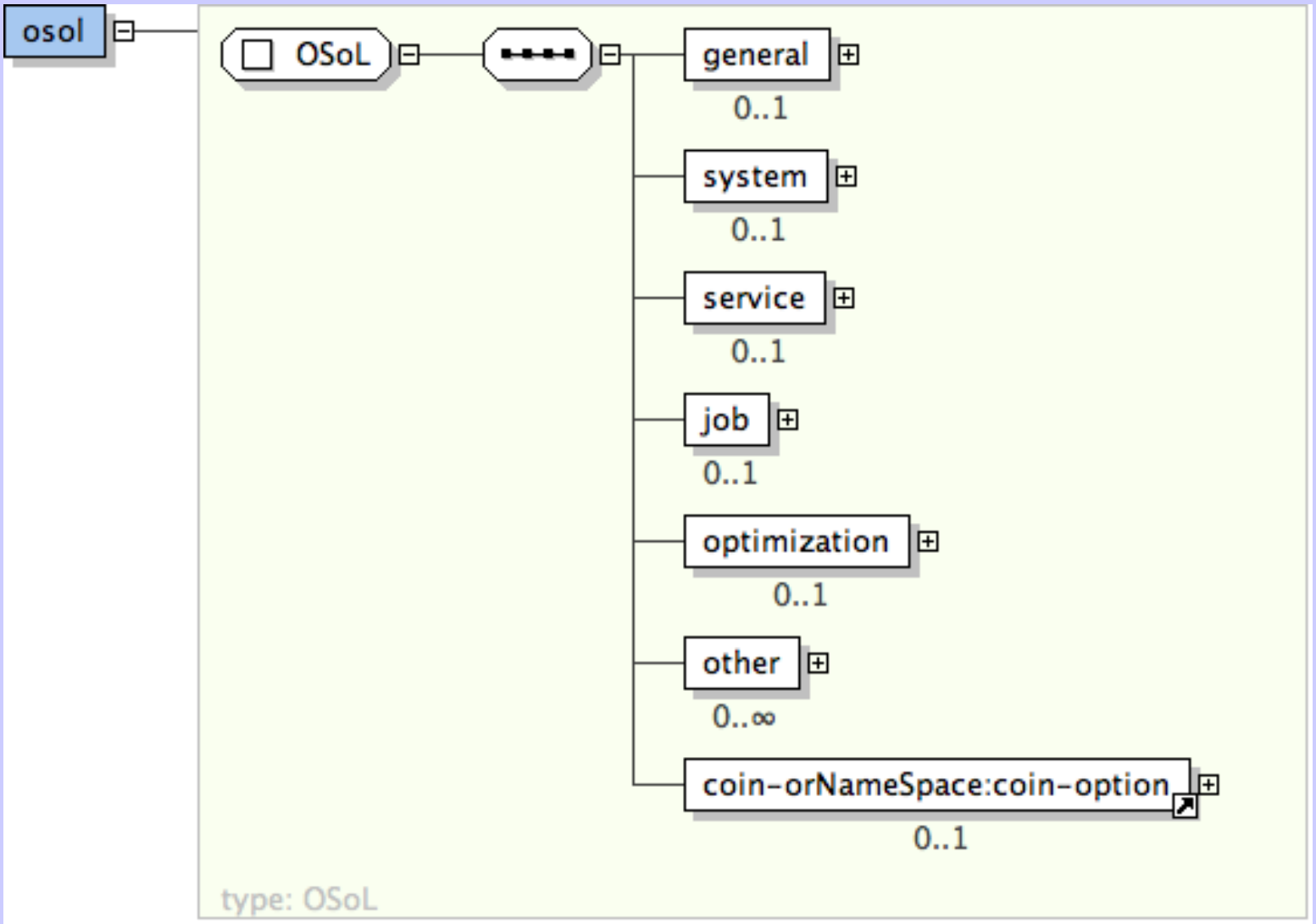
The use of the `<other>` element for constraints. Just like variables. For example you might have an `<other>` element for:

- the allowable increase
- the allowable decrease
- right hand side optimal value functions

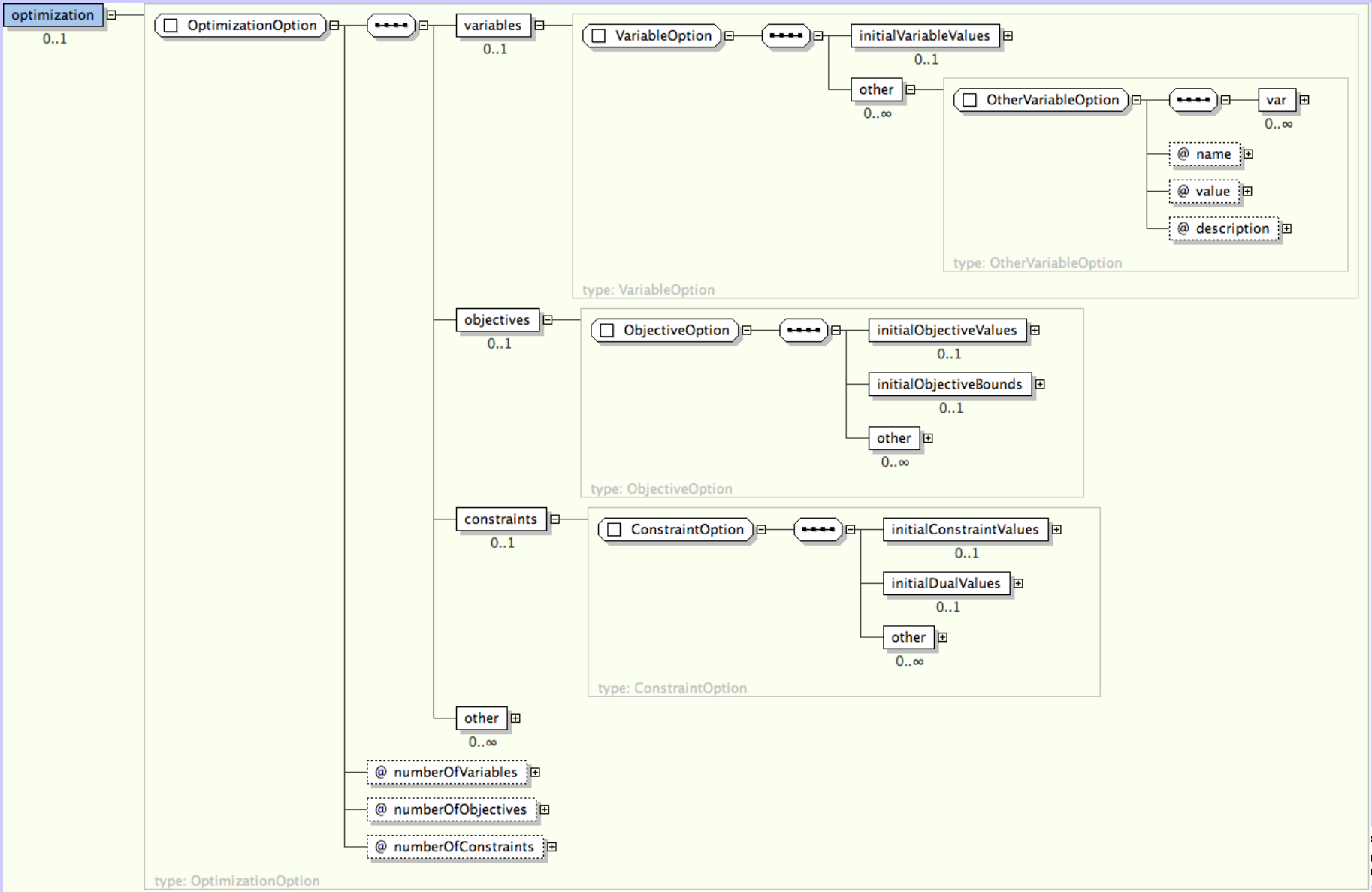
Summary: The Mapping Rules

- Each XML schema complexType corresponds to a class in `OSResult`. Elements in the actual XML file then correspond to objects in the `OSResult` class.
- An attribute or element used in the definition of a `complexType` is a member of the corresponding in-memory class; moreover the type of the attribute or element matches the type of the member.
- A schema sequence corresponds to an array. For example, the complexType `OtherVariableResult` has a sequence of `<var>` elements that are of type `OtherVarResult`.

The OSoL Schema



The OSoL Schema Optimization Element



OS Protocols: OSoL

An example of Optimization Services result Language

```
<?xml version="1.0" encoding="UTF-8"?>
<osol >
<general>
     <instanceLocation locationType="http">
          http://gsbkip.chicagogsb.edu/parincLinear.osil
     </instanceLocation>
     <contact transportType="smtp">
          kipp.martin@chicagogsb.edu
     </contact>
</general>
</osol>
```

Two important features:

- the option to have result notifications sent via email (could also ftp)
- the option to specify a problem instance on a remote machine for solution

OSoL Namespaces

A Potential Problem: there are lots of solvers and with lots of options. Naming conflicts may easily arise.

When parsing an OSoL file a solver can ignore any option it does not support. But what if two solvers have the same name but interpret the name differently?

For example, `output_level = 2` may mean different things to Lindo and Cplex.

Solution: Use XML namespaces. Sort of like an area code for XML element names.

Namespace Illustration

```
<osol>
  <general>
    <contact transportType="smtp">
        kipp.martin@chicagogsb.edu</contact>
  </general>
  <optimization>
    <other name="solverAlg">MIP</other>
    <other name="global">true</other>
    <other name="test1" description="123">thisd</other>
    <coin-orNameSpace:coin-option  name="tolerance">
      0.001</coin-orNameSpace:coin-option>
    <LindoNameSpace:coin-option  name="tolerance">
      0.001</LindoNameSpace:coin-option>
    <CplexNameSpace:coin-option  name="tolerance">
      0.001</CplexNameSpace:coin-option>
  </optimization>
</osol>
```

Summary

Design Patterns: There should be four distinct parts to the **interface** between the problem and the solver.

- a problem instance API – create the instance and get information about the instance
- a solver option interface – an instance representation should be independent of solver options
- a problem result interface
- a problem modification interface – To Do!!!